Department of Computer Science

UNIVERSITY OF BATH

Technical Report

Undergraduate Dissertation: Using Rigid-Body Mechanics And Dynamic Interaction Graphs To Create A Consistent And Believable Virtual Environment

Richard Jones

Technical Report 2004-15 **May 2004**

ISSN 1740-9497

Contact Address:

Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom

URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

Using Rigid-Body Mechanics And Dynamic Interaction Graphs To Create A Consistent And Believable Virtual Environment

Richard Jones

BSc (Hons) Computer Science

University of Bath

May 2004

Abstract

The MAVERIK graphics library (http://aig.cs.man.ac.uk/maverik/) provides a C programmer with a set of easily-portable functions and variables that can be used to create a visually impressive 3D world. It also provides routines that allow the user to navigate through the world they have created, in real time. This project seeks to take that level of functionality and add a level of physical believability – the concept that this physics system may not be a completely accurate model of the physical interactions it attempts to display, but will provide a reasonable approximation of the environment such that the user could not easily discern between the events unfolding in the system and the events that would occur in an equivalent real environment.

Although visual and aural aspects of VEs have been advancing quickly since their inception, the concept of modelling physical reality and how the world reacts to interaction has only recently begun to become computationally realistic. A dynamic interaction graph (DIG) is an extension of the idea of Bond Graphs, which involves the real-time creation and destruction of linked lists that are used to model a non-hierarchical influencial system of particles. MAVERIK is middle-layer software designed to create a 3D graphics toolkit to access the OpenGL rendering system. A solution to the problem is modelled using MAVERIK, DIGs, and basic Newtonian mechanical equations. This solution provided the user with an adequate physical simulation, but the DIG functionality was found to be severely lacking in several key regards.

Using Rigid-Body Mechanics And Dynamic Interaction Graphs To Create A Consistent And Believable Virtual Environment

submitted by Richard Jones

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of this work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed: . . . . . .

Acknowledgements

I would like to thank Professor Phil Willis, my supervisor, without whom this project would not have reached the stage it did.

I would also like to thank the family and friends who offered their support throughout the creation of the software and document.

Contents

Chapter 1

Introduction

Aim

The aim of this project was to create a single-user, real-time physics engine that can model real world environments to a high degree of believability. The key problem is one of performance, especially in large-scale VEs. To this end, the project is aiming for believability rather than absolute realism.

Objectives

- To create a library of functions and variables that provides a MAVERIK VE with physical behaviour approximately similar to that found in the real world.
- To do so at a computation rate suitable for real-time processing on a mid-range PC that supports hardware-accelerated OpenGL.
- To create a system whereby objects can be placed under and removed from the influence of other objects, quickly.
- Objects placed in this influence should be informed when any other objects under the influence are altered.

There are a multitude of tools available currently that model specific elements of physics to a very high degree of detail, but there are fewer models that provide a physical model that can be used to simulate a fully immersive and interactive environment. Such a broad overview of physical reality is needed in order to provide the user with a system that can handle a wide variety of events with a reasonably high degree of believability.

One such system is HAVOK [12], which provides an SDK designed to be used in the production of computer games. As such, it has been tailored to cater for that market, and its applications to other markets is limited. HAVOK, although it currently is, and will in the future be, among the cutting edge of physics systems for entertainment purposes, its realism is lacking when complex situations are encountered (for example, solid objects can clip into solid walls, creating an unbelievable image, and only a specific set of objects are 'physics-enabled', resulting in a lack of a sense of freedom).

This project aims to create a system that is not just for the games development industry, but for anyone with interest in designing a virtual environment that allows emergent behaviour similar to that of the real world. This means that every object in the environment must behave

as it's non-virtual counterpart would, irrespective of whether or not it has been flagged as interactable.

Products such as IOTA [9], a system developed in MAVERIK, are similar in theory to the aims of this project. However, the target of IOTA's efforts seems to be a high degree of accuracy for a low number of situations, which is precisely the opposite scope to the system being developed in this project.

As such, it is believed that this project has a high relevance to the world of physical simulation, because there seem to be few systems readily available that are of the same scope as it.

Chapter 2

Literature Review

Although it is an area of development that faces a continual struggle to be taken as seriously as it deserves, computer gaming is currently pushing the leading edge of real-time physical solutions to limits previously thought of as impossible. Specifically, the HAVOK SDK [12] has provided games developers with a fast solution that is believable enough for interactions that will not be scrutinised too carefully. Whether it is a case of supply meeting demand or vice versa is unclear, but most of the current influx of computer games contain highly-developed physical environments for the user to interact with.

While it is true that while these engines are believable in their limited context, there are still some noticable irregularities. For example, if there are only a certain number of objects that are capable of being moved, then these may be visually indistinguishable from objects that are stationary. This results in a world where some things behave as expected, and some things do not, but there is no way to differentiate between these two sets without performing a suitable experiment. It is this inconsistency that this project is aiming to avoid. By implementing collision response methods based on rigid-body dynamics, and data structures derived from Bond Graphs [3] called Dynamic Interaction Graphs, it is believed that this project can implement a believable 3D environment in which every object behaves as it would if the world were real.

If such an environment was at their disposal, then it would be an easier task for games/VE designers to create a consistent environment where everything behaves as it would in the real world. This will make the VE more convincing to the user, who will not be able to exploit any physical loopholes or find any immovable objects, all of which detract from the realism.

GNU MAVERIK

MAVERIK is a well-established toolkit for C programmers who wish to design a 3D virtual environment. It provides functionality to ensure that a convincing VE can be created with minimum effort. It provides basic types to facilitate the creation of a 3D space - vectors, lines, and tranformation matrices are provided as well as common functions for their manipulation. It also provides a series of types used to represent geometric primitives (spheres, cuboids, planes), and methods to draw these. For more complex objects, AC3D/VRML scripts can be parsed in, or a set of primitives can be joined together in a composite type.

One of MAVERIK's notable features is the ability to write object-independent callback functions, similar to C's templating capabilities. A function is written (for example, algorithms to calculate the volume of a primitive) for several different types of object, and the relevant type-specific functions are bound to the name mav_callbackGetVolumeExec() which takes the object as one of the arguments and will call the type-dependent function based on its own findings, meaning one function call needs to be written that can handle all eventualities.

There were several alternatives to using a C/MAVERIK system - OpenGL could have been used directly from C, which would provide the project with the fastest possible execution times, at the expense of having to spend a lot of time writing functionality that has already been written in the MAVERIK libraries. The J3D API for Java is a system similar to MAVERIK, providing a 3D VE API for the Java language. However, it was felt that although Java is a powerful language, its interpretive nature meant that it could not provide the real-time performance that a compiled C program would provide.

As MAVERIK is a *toolkit* for VE design, its functionality is open-ended - the MAVERIK developers could not predict how their users' VEs are supposed to behave, so almost all aspects of that behaviour has to be specifically programmed in. As such, it is a very useful tool that does not set any precedents as to how it should be used. This means that not only is it easy to add functionality to the main drawing loop, MAVERIK was specifically designed with this in mind. Extra function calls are added into the main display loop that update the parameters of the objects (position, orientation, velocity), and the display loop then draws the objects in their updated position.

Rigid Body Dynamics

On a small scale, interactions between rigid bodies are quite simple to calculate. The movement equations only contain a multiplication and an addition, meaning that even consumer PCs can easily move and display objects in real-time. However, when multiple objects are colliding with multiple objects, this can lead not only to many thousands of calculations necessary every frame, but also the introduction of rotational dynamics (introducing CPU-intensive matrix multiplication and trigonomic operations). On such a scale, even modern CPUs are going to be strained to calculate the next configuration of the system before the next frame needs to be displayed. Optimizations in this area have been

researched by the gaming/VE community for quite some time, so there is a considerable amount of background material.

a) Particle Motion

Traditionally (Euler), particles in motion are given two properties – position and velocity. Take the equations:

$$x' = x + v * \Delta t \tag{1}$$

This gives the position x' of a particle that was moving with velocity v at position x, Δt time units ago. The new velocity is calculated thus:

$$v' = v + a * \Delta t \tag{2}$$

where a is given like so:

$$F = ma \tag{3}$$

where F is the resultant force on the particle, and m is its mass. With these three equations, the path of a particle can be mapped accurately as long as the resultant force on it and a start condition (position velocity pair) are known.

However, this is not a very efficient way to do it. Why calculate the object's velocity when it is only the position we need to calculate (assuming there are no collisions for the moment)?

Verlet integration [2] removes the velocity from the equations and uses the particle's current and previous positions in order to calculate where it will be at the next time step. This means that Euler formulae only need to be calculated when the particle comes into existence and when it changes direction (collision). The Verlet calculations are:

$$x' = 2x - x'' + a * \Delta t^2 \tag{4}$$

$$x'' = x \tag{5}$$

Where x'' is the previous position of the particle. The update step is trivial, but can be further optimized by changing pointer references rather than rewriting variables.

This approach increases efficiency of the particle computation, but it has various downsides when it is combined with other aspects of the environment. For example, if the time step is too large, then a fast moving particle could conceivably pass straight through another object if the positions are not within its bounding box.

The problems associated with having motion as a discrete variable are known, and methods for counteracting the effect are presented in the section on collision detection. However, the choice to use Verlet integration should make the overall system stability higher, because only the position is known at any one time (when the velocity is needed, the difference between the last two positions will be integrated over the timestep). With Euler, both the position and velocity are calculated, which is not only a waste of clock cycles but can also lead to synchronisation problems due to floating point errors. With regards to the system specification, the equations above will give a satisfactory illusion of rigid body dynamics, although it may not be fully accurate.

The removal of velocity and introduction of a time step naturally lends itself to modelling the progression of an object through a real-time system, as opposed to the standard equations which are more suited to getting information about a dynamic system in a single configuration. Combined with the fact that this integration system has been used to a large degree of success [2] produces a considerable confidence that this method will be suitable for implementation in the system without heavy modifcation.

b) Collision Detection

MAVERIK provides its own default methods for testing for collisions between polygons. Until proved otherwise, it will be assumed that these methods are fast enough for real time particle interaction. However, alternate methods of modelling particle interaction have been researched in case they are not.

Assuming that the object has not passed entirely through the target, i.e. that at least some of their particles co-exist, then the object can simply be transformed until it is ‘resting’ on the surface [2]. From there, any further calculations can be performed (including what happens to the target and object – would they bounce away or stick to each other?) using the distance the object travelled inside the target as a basis for calculating the force that went into the collision.

The Verlet integration scheme creates some problems when testing for collisions (as it is a discrete function), but there are a number of ways to get around this:

i) Vector Projection

Vectors can be projected from points on the object between the two positions. In three dimensions, this can cause problems (where should rays be cast from in order to ensure all collisions are detected?). These choices can be narrowed down using bounding boxes to perform a crude check to see what area the collision occurred in, and then using more thorough methods over a smaller data set.

ii) Cylinder Projection

Cylinders are a fast primitive to calculate, so a capped cylinder of suitable radius could be created between the midpoints of the object at both iterations. If the cylinder clashes with an object, then there was a collision.

However, both of these ideas can cause problems if the object is arcing (i.e. has a motion in the x axis and is accelerating negative y due to gravity), as the projections will be straight lines. A method to circumvent this problem would be to use a projection method to get a time of alleged collision, and then use Euler integration to get the object's actual position at that time. If there is a discrepency that places the object outside of the collision zone, then perform another projection from this point, and so on until the object a) reaches the point it did through Verlet integration, or b) collides with another object.

An alternate way of solving this problem would be to cast particles themselves instead of rays and cylinders (i.e. primitives that are themselves subject to the physical laws of the environment). If the particles are cast using Euler integration, then the amount of non-Verlet computation used is minimised, leading to an overall gain in speed, but the accuracy of the system configuration at the moment of collision is maintained. This will be considered should there prove to be any problems with the scheme mentioned above.

Of course, most of these problems can be removed by having a small enough time step interval.

c) Collision Response

Once the collisions have been detected to within a certain tolerance, then the outgoing velocities of the bodies involved must be calculated and applied. This process is the collision response, and together with collision detection it is the most demanding on CPU time. Rigid body dynamics need a special force in order to account for the high level of velocity change (both magnitude and direction) that takes place in a small amount of time, termed the impulse. The impulsive force acts in the direction of the normal to the collision, at the point of collision. As the direction is known, all that is needed is the magnitude (the direction will be reversed but of equal magnitude for the other object in the collision), and this is calculated via the following equation (taken from [4]):

$$j = \frac{-(1+e)\mathbf{v}_1^{AB} \cdot \mathbf{n}}{\mathbf{n} \cdot \mathbf{n}\left(\frac{1}{M_A} + \frac{1}{M_B}\right) + \left[\left(\mathbf{I}_A^{-1}(\mathbf{r}_{AP} \times \mathbf{n})\right) \times \mathbf{r}_{AP} + \left(\mathbf{I}_B^{-1}(\mathbf{r}_{BP} \times \mathbf{n})\right) \times \mathbf{r}_{BP}\right] \cdot \mathbf{n}}$$

(6)

A and **B** are the objects in the collision, e is the coefficient of resitution, $\mathbf{v}_1^{AB}$ is the resultant velocity of the objects, **n** is a vector of the normal to the collision, M is the mass, **I** is a 3x3 matrix defining the inertia tensor [7] of the object, and **r** is a vector from the centre of mass of the object to the point of impact. J is a scalar that should be multiplied by **n** to give the impulsive force applied in the collision. This is then divided by the mass to give the acceleration, and this is differentiated to give the velocity of the object after the collision. Likewise, this scalar can be applied to a vector perpendicular to both the vector from the centre of the object to the point of impact, and the velocity of the object at the point of impact in order to give a value for the torque of the object after the collision, and this can be processed in a manner similar to the linear velocity in order to give the outgoing angular velocity.

d) Frictional Interaction

The introduction of the DIG concept creates the possibility of interacting forces between objects that are in constant contact, such as friction. When an object in a DIG is moved, checks should be performed to see if it should have influence over any other objects in the DIG (for example, a ball on a table will not influence the table when it is pushed), then the resultant forces may be calculated.

In the Coulomb frictional model [6], the frictional force between objects is proportional to the force exerted by the surface normal on the object resting on it, which is in turn proportional to the mass of the object. This makes sense – a heavier object is going to need more force to move it than a lighter object of the same material, when resting on the same surface. In addition, a greater force is required to start an object moving (static friction) than is required to keep an object moving (dynamic friction). Equations to demonstrate this are given in [1], and these are simple enough to be included without further optimisations.

e) Deformable Bodies

In essence, these are introduced when the DIG system is used on the particle level. If the strength of the binding between two particles is not infinite, then there will be a delay between one particle moving and the others following. This could be used to model soft-body systems, such as cloth (outlined in [2]). However, in order to ensure that the system always behaves as it should, the number of particles needed will grow exponentially (in order to model a spongey sphere, for example) unless the system is trivial (Bridge Builder's struts).

As such, these aspects of physics will be implemented for trivial systems only – [2] gives a brief explanation of models that could be used in the Miscellaneous section, which involves relaxing the constraints between two particles contained in a DIG so that the particle that is not directly being influenced will not attempt to snap itself instantly back to the 'correct' (rest-length) position, but take a logarithmic approach. This gives the system a sort of 'lazy' feel, and the speed at which particles snap back can be altered depending on the material.

f) Destroyable Bodies

Impacts are simple to model: a force proportional to a plank of wood (anchored at both ends) would cause deformation (bending) in the direction of the force. If the displacement distance oversteps the material's elastic limit (a constant member of the datatype, initialised at creation time), then it will snap. Forces in the middle of the wood will create more displacement that forces at either end.

In the context of being able to weaken previously solid structures (sawing partially through a table leg, for example), this could be implemented by reducing the structure's elastic limit variable at the point of impact by an amount proportional to how much it has been compromised.

These systems would both be useful for creating emergent behaviour (if every object in an environment is breakable, which is mostly accurate in our own environment, then all sorts of solutions to problems can manifest themselves), and so it should be implemented. However, this is one of the least important aspects to physical realism (in fact, most games have no concept of anything being destroyable other than enemies), especially when the overhead involved is taken into account.

Dynamic Interaction Graphs

Physical modelling is a computationally demanding application, and as such, unnecessary instructions are to be avoided wherever possible. If an object near the bottom of a stack of objects is moved, then all other objects in the stack could be affected. If the total number of objects in the stack is a small portion of those in the total VE, then it is inadvisable to poll every object's position to see if it is located in the stack. To this end, it is much more computationally sound to keep a record of which objects are in the stack and traverse this list (adding or removing objects as necessary) instead. The larger memory footprint left by such a structure would be insignificant when compared to the gain in the scalability of the program.

The DIG approach can also be adapted to a smaller scale by using it as the basis for creating a multi-particle object. Using Bridge Builder as an example, consider each of the struts as a

DIG containing two objects, being the particles at either end. In this case, the link between them is both visible and has physical properties. However, on the basic assumption that objects cannot be altered from their original configuration, then this graph would remain the same throughout the simulation process, giving rise to the definition of a Static Interaction Graph.

There are issues with this approach – objects that are not in contact with each other but are in contact with a floor should not be part of the same DIG (assuming that the floor is fixed and not subject to the laws of mechanics other than in the way it influences objects that contact it). This is done for many reasons –

- The floor cannot move, and so all objects in contact with it will not need to be polled at the same time.
- Changes in objects on the floor should not affect objects that are on the floor but not contacting the changed object.
- If the system has gravity, then after a finite time all objects will be linked to the floor, and so the use of a DIG would be equivalent to polling every object in the system.

The main application of a DIG is one of time-saving – if a table with objects on it moves, then rather than having to poll objects in the system in order to ascertain which are in contact with the table and have to be altered, the relevant force can simply be applied to all objects in the DIG of the table, and the end result will be achieved in fewer CPU cycles.

Contemporary Engines

The concept of modelling physical reality has become a very important aspect of gaming/virtual environment technology. The advances made by the gaming industry in creating believable visual and aural environments have been sizeable when compared to the relatively small steps taken in the creation of physical environments. This could either be due to an increase in the sophistication of the average consumer, or the result of the industry taking advantage of the continual increase in the average processing ability of a mainstream PC. However, few engines have achieved an environment that is completely believable either in its consitency or its non-determinism - scripted events are usually used as a replacement for behaviour modelling, leading to endless repetition that removes any illusion that the environment is analagous to our own.

The HAVOK SDK [12] has recently received press coverage for its use in a significant percentage of games engines that require a realistic physics environment. So much so, in fact, that the name HAVOK has become almost synonymous amongst gaming circles with believable physics engines (in forum request lists for game sequels, members are requesting the use of HAVOK in future editions of a game). Immersive physics environments are now important to the success of a game to the extent that the inclusion of one can be an incentive to purchase it.

HAVOK is mainly used in the modelling of physics in first-person perspective games, and the actions that typically occur in these games only provide a limited cross-section of rigid body dyamics. There are other aspects of physics that are modelled in isolation by games using proprietry engines, for example Bridge Builder [11] handles stress and strain of struts used in the creation of suspension bridges as crossed by a heavily-laden train; and Truck Dismount [10] provides a highly-customizable engine that is only possible through the accurate modelling of the physics involved – any scripting would severely limit the ability of the user to affect the process.

It is not possible to test the HAVOK SDK itself, because the source code is not available to non-developers. However, it is possible to perform black-box tests on the games that have incorporated the HAVOK physics system in order to ascertain an idea of what sort of physical modelling it provides. The two games specifically mentioned above can be examined in a similar manner - no source code is available, so any testing will be purely black-box.

a) HAVOK [12]

Games that use the HAVOK SDK have a reasonably consistent physical system in operation. However, as the games themselves vary, the information that can be gathered is not consistent. There are constants, however – even though a physical system is in operation, the developers need to set specific objects to be 'physics-enabled', otherwise user interaction will yield no results.

Deus Ex 2 [13] is a useful example of what can be achieved in the HAVOK engine because it includes the ability to pick up and manipulate objects. Complex object placements are valid – early on in the game, the player encounters a storage unit containing a stack of cylinders on their sides, held in place with a single plank. If the plank is moved, the barrels roll out in a convincing (non-scripted) fashion. Throw an object at the top of a fence so it doesn't quite clear it, and it will spin around while still keeping most of its momentum. This demonstrates a huge step towards the presence of an immersive environment for games.

However, limitations are present - objects the player is carrying will not occupy any physical space (or affect mobility) until it is dropped. Objects that are picked up are not removed from the external environment until all the objects above it have fallen to rest, breaking the illusion of their solidity.

Half Life 2 [14] uses a heavily-modified version of the HAVOK SDK, codenamed Source by the programmers responsible. To demonstrate how detailed the modified HAVOK engine is, [15] mentions a demo room created by the programmers where a huge-scale, accurately modelled engine was created. A large wrench was supported

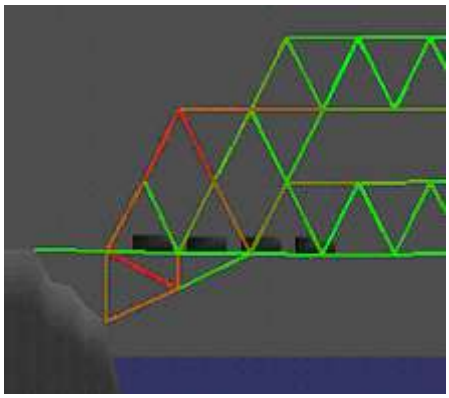
over the piston chambers such that if the supports were shot away, it would fall and cause the engine to grind to a halt. This was not scripted in any way, it is just an example of emergent behaviour possible because of the level of effort that went into the engine design.

An interesting aspect of the Half Life 2 level creation process is that textures are no longer just added to an object to give it visual character; each texture has a range of physical properties assigned to it, so a wood texture will make the object behave and sound like wood. This simple idea has repercussions in the area of combining the visual, aural and physical aspects of an in-game object, making it much easier for a designer to create a believable environment. If this could be applied not only to textures but to physical objects, then a library of standard templates could be created and called upon when needed. This would make the creation of consistent environments much simpler for developers, and the results more immersive for users.

b) Bridge Builder

This simple 2D game involves the user attempting to create a bridge over a ravine. Two objectives are included in that synopsis – the bridge must be under budget (each strut costs a fixed amount), and it must survive the passing of a heavy train load. This approach simulates two particles for each strut and stores a rest length for them. When the train passes over, a force is applied to one of the particles, and the struct continuously uses its internal force to try to regain its rest length. This process is iterated through the bridge model, such that the train presses on a particle, which presses on its strut, which causes the other particle to exert a slightly lesser force onto the struts it is in contact with, and so on through the structure. The struts have a hard-coded strength constant which means it will snap if the force exceeds a certain amount.

This approach can be adapted when considering multi-particle objects that are moved by the environment – the force only needs to be exerted on one particle, and it will direct the force through the rest of the system according to its properties (elasticity, strength). This has the added advantage of giving a simple method of implementing rotation (two opposite forces acting at opposite sides of an object).



Bridge Builder. Green-red displays a range of stresses, the black blocks are the train carriages.

c) Truck Dismount

Another simplistic game created around a convincing physics engine, the premise being to cause the most damage possible to a rag-doll humanoid by placing him between a wall and a high-speed truck. There are ramps that can be moved or removed, demonstrating a good application of constraint-bound and impulsive physical methods. The physics are very believable, and the game copes well with real time rotations, inelastic collisions and a complex particle system (the character slumps in a convincing humanoid way).

Control panel for Truck Dismount

L-R, Top to bottom: Z position of character, Z positions of ramps, position of character (1 – 6), Windscreen toggle, Acceleration of truck, X position of ramps, ramp toggles.

One of the most impressive things about Truck Dismount is the wealth of options available to the user in order to configure their dismount. This shows what can be created by designing a game from the engine-upwards – by changing a few values

slightly, the results can change immensely. The amount of customisation means that no two runs of the truck are the same.

This is clearly a sign of a well-crafted, robust engine, as it shows that every feature is designed with flexibility in mind – there are no hacks in order to fix a single solution, and no scripting, otherwise there would be no possibility for other solutions.

Summary

In order to create an immersive environment, the following possibilities should be accounted for:

- Static and dynamic particles (linear, exponential and rotational dynamics)
- Friction between objects (static, dynamic and rolling friction),
- Imposed hierarchies of objects under each other's influence,
- Stress and strain of multi-particle systems,
- Particle/object collisions,
- Destroyable objects

As has been outlined above, there are widely-known methods for the application of these aspects of physical environments. Respectively, these are:

- Verlet integration,
- Coulomb friction model,
- Bond graphs/DIGs,
- Infinite-stiffness springs,
- Bounding-boxes and ray intersection,
- Displacement and elastic limits

The Coulomb friction model should be simple enough to implement that it does not require further optimizations in order for real-time operation. As opposed to particle motion which is used any time any particle is moving, this system only comes into effect when two objects are touching and there is a non-zero resultant force. Not only does this take up a small portion of the total simulation time, but the necessary computations are trivial when compared to others that are performed on a much more regular basis.

The DIG approach to modelling objects in contact has been performed to a large degree of success in [5]. Although the subject area there is small, all that needs to be done behind the scenes when two objects come into contact is two pointer updates. However, there is a reasonably sizeable overhead that needs to perform checking on the messages passed in order to ensure that the same cog is not receiving a message it has already processed. If there is an

implementation that could perform the same task without needing the checking overhead, then it would be a preferable method for this project.

Infinite-stiffness springs will be used to model the links between particles in a rigid body. Using this system, a force can be applied to one of the particles, and the rest will alter their position in order to keep the body's shape. The infinite-stiffness means that this change happens almost instantaneously, with little or no visible deformation. Another advantage of this model is that only one force needs to be applied on one particle, which should reduce the computation time.

Particle collision is one of the most intensive and regularly repeated methods necessary. The number of times this operation needs to be performed can be minimized by using well-shaped bounding boxes in order to only perform more computationally intensive checks when the probability of a collision is very high. If the bounding box check reveals a collision, then the cylinder/ray casting method or the Euler particle method will be used, depending on the movement characteristics of the particle (linear velocity – ray/cylinder intersection; accelerating – Euler particle).

Displacement of a multi-particle system should give this physics engine a degree of realism not seen in widely available engines. It will allow for destroyable supports (within reason), and isn't very processor intensive – however, it does lead to particle motion and collision, and the hierarchies of objects will need to be changed. Partial weakening of materials can also be modelled with minimal computation time given the right data structure (the information stored will need to have a factor of weakening and a position, as well as the original material's strength).

Using the numerous shortcuts detailed in [1] and [2], it should be possible to model most of these aspects in the MAVERIK graphics library on a reasonably modern PC in real time. However, the graphics quality will be questionable because time will not be spent on it. Also, there will be no audio to speak of, so the engine will be allowed more CPU time than it would be in a standard gaming/VE operating environment.

Chapter 3

Requirements

A software development project is most likely to succeed with a clear and well-defined set of requirements. However, a software development project usually has a specific user base towards which it is tailored, and from which requirements can be gathered. This project is being created mainly for research purposes, as such the requirements are academically driven – this project is a test to see if the theory behind it is feasible on today's hardware.

Requirements Analysis

The requirements-gathering process was conducted mainly through a scenario based approach, as documented in [16, section 6.2.2]. This approach was used because knowledge of how real-world objects behaved in a certain situation directly led to information on how objects should behave in the simulation system. Requirements based on the DIG generation were implemented after examining [5] and [17].

The project synopsis provided in Appendix A laid out guidelines for the project, and this provided us with an overall scenario to be created, and some requirements were harnessed from this.

Requirements Specification

The following chapter is based on the IEEE specification for writing an SRS [18]. As well as defining the operating environment and target userbase for the system, this section orders the main functional requirements into a two-tier system of primary requirements (that the system must meet or be considered a failure) and secondary requirements (which will increase the system's success but are not necessarily going to be implemented, or their implementation will prohibit the success according to the primary requirements).

Functionality

This project's main objective is to create an extension to the MAVERIK library that adds a level of physical behaviour that would otherwise be absent. This objective must take the following into account:

- Real-time execution on a readily available PC
- Methods providing support for collision detection and response
- Dynamic creation and destruction of object groups
- Emergent behaviour that is not specifically programmed in

The program must be created with the knowledge that believability combined with real-time operation is more favourable than absolute accuracy combined with slower operation.

External Interfaces

Output

The software interfaces with the system's hardware through the MAVERIK library, which interfaces with the GL implementation on the machine. This provides a degree of portability that means the system will function on any machine with a functional MAVERIK distribution.

Input

It would be desirable for the program to be able to apply its physics layer to a variety of 3D landscapes that are provided as input, but this is not a requirement at this stage of the development cycle. The most important thing at this time is the creation of a framework that can handle physical reality for a pre-programmed set of objects; this framework can be extracted and made standalone at a later date. As such, the user has a very limited role to play in the input to the program, and output is provided through MAVERIK's standard graphical output methods.

There are no input requirements other than that there must be sufficient data present in the simulation to confirm the fulfillment of the requirements. This will be achieved by providing a series of demonstration programs that allow a non-technical user to examine the capabilities of the system.

Target User

As this project is primarily for research purposes, the mass-market potential is quite limited. As such, the only documentation provided will be this document, and the target user is assumed to be the marker of the project. The user interface is a very low priority, although there should be some degree of user control in a test application (dynamic object creation, environment navigation, etc.).

Target Environment

The primary speed aim of this project is to achieve an acceptable frame rate on a mainstream desktop PC, equivalent to those found in Bath University's library. The PC must be equipped with a functional MAVERIK distribution (the project was compiled against version 6.2), and a method to compile the source code. A statically-linked version for Windows environments will be provided for the ease of the marker.

Features

The software has two primary requirements, without which the system should be considered incomplete with respect to this document. If the system can provide evidence on a collision detection/response system, together with a method for imposing peerless hierarchies (DIGs) on objects, then these requirements are fulfilled.

There are a number of secondary requirements that, while they are not necessary for completeness, their inclusion would mean that the system can provide a coherant and believable model for physical reality to a very small degree of error. These include frictional modelling (as opposed to a scalar division of velocity, which would be an inaccurate method, but provide similar results to a user), deformable/destroyable bodies, and the creation of multiple-particle systems acting as one coherant body.

NB – the multiple-particle systems are not analagous to a set of objects linked together in a Dynamic Interaction Graph; the links between particles should remain unless the body is destroyed, hence Static Interaction Graph.

Primary Requirements

Collision Detection/Response

The system must include the functionality to detect whether or not a collision has occured between two physical objects using logical quantifiers based on the distance and size of those objects. The area of polling for the collision detection should be limited to those objects whose bounding volumes are currently overlapping, which should be calculated either using MAVERIK's built in bounding box methods or otherwise.

MAVERIK's bounding volume creation methods are axis-oriented, meaning that if an object is not in a standard orientation, the volume returned will be larger than the volume the object

actually occupies, even in the case of rectangular parallelepipeds. The alternative is to generate object-oriented bounding volumes, but this will limit the speed at which intersections can be calculated due to the rotation operations necessry in creating/manipulating the volume.

Although there is no mention of collision detection/response in the project specification, it is only through the creation of these aspects of an environment that emergent behaviour can be nurtured, and that was a specific requirement.

DIG manipulation

The system must include the functionality to create and destroy links between objects when it is deemed that they enter or leave the influence of each other. As opposed to [5], where cogs were placed in a limited grid layout, and items in physical contact were known to be extering influence over each other, the application of the DIG idea to a full 3D environment with six degrees of freedom is considerably more complex. Physical proximity should certainly be taken into consideration when calculating DIGs, but it is by no means the only factor. The resultant velocity between two objects must also be used, and the relative positions – for example, a ball next to the leg of a table at ground level should be treated differently to a ball lying on the table-top (of course, no objects in the environment will be designated as a table-type, this is just a figurative example).

With an assumption that the gravitational force is acting negatively in the Y axis (that is, a constant acceleration [0, -9.81, 0] is being applied to all objects), then the vertical positioning of objects will be a factor in the determination of DIG layouts. However, it is important to define the phrase ‘DIG layout’ – the DIG should be a peerless system, as there will be no information stored such as “A is on top of B” if A and B are in the same DIG, just by looking at the DIG. All that will be known is that A and B are under each other’s influence.

An important observation is that, in impulse-based physics systems, there is a tendency for objects not to rest on a surface when it looks like they should, but to bounce on the surface with ever-decreasing strength. As such, it is necessary to implement contraints that force an object to have zero velocity perpendicular to the surface when the impulsive force is sufficiently small as to be negligible. When this constraint has been imposed, however, it effectively removes the object from the list of objects that have physical characteristics, as collisions are no longer being resolved. As such, the object must be specifically notified when its position needs to be updated. At which point, the object will no longer be at rest relative to the surface normal, and so it rejoins the set of objects that are behaving normally. The point of the DIG system will be that objects will rejoin the set of physical objects when any objects that it is touching are moved. If, after this event, the object again has a negligble impulsive force, then the cycle can be repeated. However, it is of crucial importance that objects are seamlessly tranformed between being constraint-based (at rest relative to a surface’s normal) and impulse based (in motion).

Secondary Requirements

Friction

The Coulomb friction model provides three constants for any surface, which must be determined through experimental evidence, they cannot be calculated [6]. The constants (static, dynamic and rolling constants) are multiplied by the normal force acting on the object from the surface, and this gives the magnitude of the force that opposes motion, which acts against the direction of motion (or the force that is attempting motion if the object is still). Static friction is applied when the object is stationary relative to the surface, dynamic friction is applied when the object is sliding relative to the surface (and is lower than static friction), and rolling friction is applied when the object is rolling over the surface. The ability to apply these forces accurately implies that the overall velocity of an object is calculated via the summation of the forces acting upon it, which is not necessarily the case.

This is not a primary requirement because it is possible to simulate the effects of friction. By continually degrading an object's velocity by a constant scalar when it is contact with an object, a decrease in speed is produced which is equivalent to the kinetic energy being lost to heat and sound that friction caters for. This enables the simulation to be believable without the overheads produced by the extra frictional calculations.

Destroyable Bodies

An aspect of realism often overlooked by games/VE engines is that objects have a finite capability to resist forces. However, it is with very good reason that this functionality is not present/very limited in current engines – calculating the change that occurs in an object when sections of it are removed, in real-time, is a very demanding task that may even be impossible without taking large liberties in either the calculation of how to divide the object, or where the object parts should go after the division.

As is mentioned above, this requirement is most likely going to be impossible to fulfill without sacrificing believability by implementing scripting. As such, this is not a primary requirement, and is not expected to be implemented to any degree.

Deformable Bodies

Related to the Destroyable Bodies section, the addition of an ability of objects to be deformed from their rest state (remeniscent of the struts in Bridge Builder [11]) would add to the realism of a simulation – add a large amount of weight to a horizontal plank supported at either end, and it should sag in the middle. However, it would be conceptually impossible to

implement this without some functionality provided for the destruction of bodies – if the force making a body sag continually increases, then the displacement of the object is going to increase past the point of plausibility – as the destruction capability will probably not be implemented, this could have the effect of functionality for extra realism actually making the simulation seem less realistic. For this reason, this is not a primary requirement of the project.

Static Interaction Graphs/Multiple-Particle Systems

As opposed to Dynamic Interaction Graphs whose layout will change during the course of the simulation as objects are disturbed and conjoined, Static Interaction Graphs map the links between the parts of a multi-particle/multi-object system that should not change during the normal operation of the simulation (unless the body is deformed/destroyed). The SIG will need to hold information relating to the strength of the bonds between the particles and a rest length – the rest length is the distance between the particles when no forces are acting, and when the distance between the objects is changed, a correcting force will be applied that is proportional to the bond strength.

This is not a primary requirement because the addition to the system of the capability to produce multiple-particle bodies will not have an impact on the system's ability to fulfill the project specification.

Chapter 4

Design

MAVERIK imposes a small number of restrictions on to the programs that implement it – in order to register a new class with MAVERIK, as this project does, the callback functions must be written and registered before the library is initialised; also, there must be, at the end of the **`main()`** function, an infinite loop in which the methods for drawing frames are included. The use of an infinite loop causes some restrictions regarding which design methodologies could be implemented.

Unit Decomposition

The design of a system can be easily partitioned into a set of smaller systems with compatible interfaces [16, section 10.3]. To this end, the requirements were analysed and the following set of components were identified:

- Collision detector
  - This unit must poll every object in the environment in order to check if its physical constraints are being infringed upon by another object.
- Collision resolver
  - The resolver must take information from the detector concerning what objects are involved with a collision, and then attempt to reach a satisfactory conclusion as to what positions and velocities (linear and angular) the objects have.
- DIG alteration
  - The unit with the responsibility for creating and destroying links in the graphs must take information from the collision resolver and determine which object's links are forged, which are broken, and which are unchanged in the frame.
- Graphical output
  - The graphical output methods will be handled by MAVERIK's interface to the operating system's underlying GL implementation. This reduces the complex prodedure of drawing a set of 3D objects to a minimal group of function calls, allowing resources to be allocated to the units applicable to the project specification.

Given the information above, it is possible to formulate a diagram indicating the program flow.

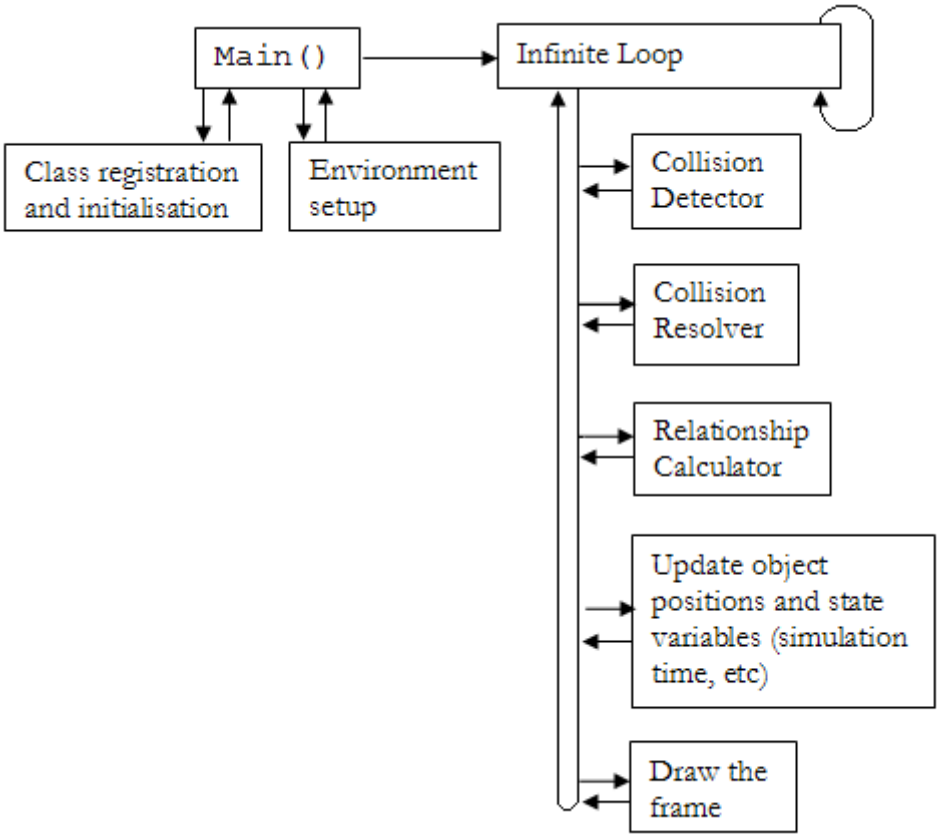


Figure 1: Program Flow

Physical Data Structure

The physical information that must be held concerning an object can be divided into four parts: Data that MAVERIK itself holds on objects; constants; state data; and debugging information (also useful to control program flow).

Object Data

The physical wrapper is going to have to hold data on the type of object it is holding, although it need not be explicitly defined. A `MAV_object` datatype holds pointers to two variables, one called `the_class`, which points to a constant datatype indicative of which type the objects is (e.g., in a sphere object it would point to `mav_class_sphere`) and another called `the_data`, which is a pointer to a specific type of structure (e.g., for a sphere object it would point to an instance of `MAV_sphere`). The type-specific structures contain information regarding the size of the object which varies from type to type (`float radius` for spheres, `MAV_vector size` for boxes), but there are a number of constant members such as `MAV_matrix matrix`, a 4x4 homogeneous transformation matrix that describes

the object's current scale factor, translation from local to world co-ordinates and a 3x3 orientation matrix. It is this matrix that the draw callbacks examine when they attempt to draw an object, so this matrix will need to be updated with the state data from the physical wrapper.

Constants

`float mass`

Needed to calculate the momentum when entering a collision, and when resolving forces on an object.

`Float MuS, MuD, MuR` or `float Restitution`

Frictional co-efficients for static, dynamic and rolling friction if the Coloumb fictional model is to be implemented; otherwise a single coefficient of restitution between zero (all energy is lost on collision) and one (no energy is lost) can be applied to the velocity on every collision in order to have total system energy tend to zero.

`MAV_vector CentreOfMass`

The position (in local co-ordinates) of the centre of mass. For all MAVERIK-defined primitives, this is the vector [0,0,0].

State Data

```
MAV_vector prevPos, currPos, nextPos,
           prevRot, currRot, nextRot
```

Three vectors for position and rotation allow Verlet integration to be used. When velocity is needed, then the difference between the current position and the last position can be taken. When the module to update the positions is called, is can copy the current position to the previous position, the next to the current and then the current position into the tranformation matrix's translation components (for linear change), or create an updated rotation matrix (for angular change).

`MAV_vector ResultantForce, Torque`

These vectors hold the forces that are acting on the centre of mass in both linear and angular directions. These vectors do not have to be constantly maintained, but only altered when an object is in a collision.

`MAV_vector Velocity, Omega`

The linear and angular velocity of the object. Differentiate these with respect to the timestep in order to find out the distance the object travels in a frame update.

Debugging Information

It is unknown at this stage what sort of extra information will be needed by the physical wrapper in order for it to function correctly. However, it can be theorised that some sort of object identification number will be held in order to identify what objects are where in the simulation. If messages are used to communicate collisions, then information on the last message received (identification number, time, where it was sent from) would be helpful in ensuring the same collision was not processed more than once.

Class Registration

The first stage to the system is to register the class that will be used to hold the physical data of the MAVERIK objects. This includes the definition of the type, the registration of this type with the MAVERIK library, and the assignment of several callback functions that MAVERIK will use to perform display tasks on the objects. These callback functions include descriptions of how MAVERIK should draw the objects, how their bounding boxes should be calculated, and so on.

The simplest way to implement these would be to simply execute the callback function for the object that the type holds. For an object holding physical characteristics of a sphere, for example, then the draw method for a sphere can be called when it needs to be drawn, with no adverse side effects. This approach is also scalable – a composite object can simply iterate over the objects it holds, calling each of their draw callbacks in turn (however, some form of Z-buffering would be necessary in order to make sure that the objects further away from the camera do not appear in front of those that are closer).

Environment Setup

In order for physical reality to be tested, there must be some form of environment available to the program. The simplest form of which that provides very little room for erroneous behaviour would be a rectangular area with six walls in order to constrain movement on all sides. The removal of any one of these walls would allow for an object to ‘escape’, probably leading to floating point errors and other unwanted behaviour as it falls towards an infinity point.

Another way to ease the testing and debugging of the physical algorithms would be to constrain all objects created to a 2D plane. If, for example, the Z co-ordinate of each object

was equal, then the motion would be easier to visually monitor. If the algoithms are created without this constraint in mind, then the scalability will not be affected as the constraint is relaxed.

Main Loop

The main stage of the system's execution involves repeatedly calling a small group of methods with the same data (the set of physical wrappers, and information on the DIGs). As neither encapsulation nor security are requirements (due to the non-networked, single-user operation), the variables can be global scope. This will remove any concerns that functions will be receiving different sets of data. Also, if the data is stored globally then there is no need to pass them as arguments, or for the fucntions to return values. This will manifest itself as a small speed-up in the function call execution time.

Collision Detection

A large factor in the overhead of a physical system is that every frame, every object must be checked against every other object for physical interference. This means that, if n is the number of objects in the system, there is an $O(n^2)$ complexity operation being performed every frame, even before any calculations relating to impulse application are performed.

This unit must contain the functionality to pass on information concerning which objects are involved with a collision this frame. As was used in [5], a message queue system has several advantages over other methods, such as flags built in to the physical wrapper. Primarily, flags will have to be initialised and re-set after they have been read, which could lead to strange behaviour if the programmer is lax in these issues. Also, the infrastructure for a suitable message processing system and a class defining the object specification by the author of [5], which could then be integrated into the simulation system with a minimal expenditure of effort. Additional functionality could be added to the message objects (such as a point of impact vector), and it helps to define the boundary between the collision detection and resolution units: The collision detector has control up to the point of sending the message, then the message processor delegates to the collision resolver if there are messages in the queue.

This unit may also have to apply boundary constraints to an object (that is, check each object's bounding box to see if any part of it lies outside the global environment constraints defined statically), and update the positions accordingly. This process is relatively simple (translate the object parallel to the surface normal until they are no longer interfering), and as such may not require the dispatch of a message indicative of a collision event.

Collision Resolution

When the detection phase is completed, the message queue must be parsed in order to find out if there are any collisions. Information in the message should include which objects are involved with the collision, and the point of impact. Using this information and information that is globally available, it should be possible to calculate the outgoing velocity vectors of the objects involved.

In order to calculate this, it must take information from the objects concerning their velocity as they enter the collision, and their relative positions. Given that the message object contains references to the objects involved in the collision, this should be easy to obtain. The impulse force can then be obtained through equation (6), and if this is significant, it should then update the information contained in the physical wrapper, including the outgoing velocity.

If the force is negligible, then operations must be performed in order to check whether or not the object with the negligible escape velocity should be added to the DIG of the other object in the collision. The nature of these operations will depend on the nature of the objects in the collision – is one stationary? Where is the object with negligible force in relation to the other object? The system will need to calculate answers to these questions in order to reach a satisfactory solution.

Simultaneous collisions can be handled by implementing a check on the time of the last collision message received – if there was another collision resolved in the same frame as the pending collision, then the initial velocity will be in the object's velocity member, and so it should be taken from there and not through Verlet integration of the object's previous positions.

DIG Alteration

After the collisions are resolved, there may be a change in the graph status of some objects. For example, if an object is bouncing off the top of a flat surface with a very low impulsive force. In this case, the objects should be bound together in a graph (given the constraints outlined in the Literature Review).

If there is any change in the link status, this may in turn affect the position of the objects involved in the next frame – this may result in the collsion resolver needing to be re-called, or the DIG alteration unit may have in-built methods for these kinds of collisions (it is unlikely that this unit will need to calculate a collision impulse, and as such the complexity will be low).

There are many methods suitable for storing the relationships between objects. [5] uses the objects themselves – one of the members of the structure is a pointer to an array of objects that contains each object that is directly influenced by the object in question (the object's 'neighbours'). This is a suitable method for the referenced system, mainly because there is a maximum of four objects in a neightbours list due to the functionality of the program. A limitation in this method is that objects in the neightbours list of one object will have their own neightbours list that will contain some repetition. This is an undesirable feature because of the avoidable overhead in polling the same object more than once.

The method that is currently being considered for this task involves a global scope two-dimensional array constrained in both dimensions by the maximum number of objects that are allowed to be present in the simulation at any one time. This array will contain some method of referencing an object, either by it's debug ID or a pointer to the object itself. This method is clarified with the following diagram:

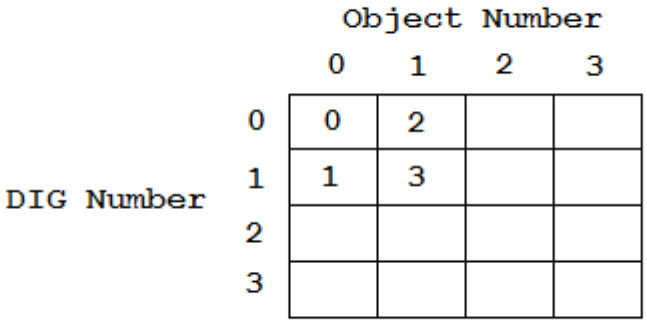


Figure 2: Example Graph

In this example, the maximum number of objects possible (and thus the constraint on the DIG array) is four, objects zero and two are in the first DIG, and one and three are in the second. DIGs two and three are empty and unused.

A side-effect of using this method for DIG manipulation would be that objects that are not interacting with other objects could be placed in a DIG with no other objects. This would aid the scalability of the system, as functions can accept a DIG as an argument and perform processing on all of the objects within regardless of how many there are. This will need to be decided on a function by function basis, however, as it may not be suitable to process every object in a DIG as if it were not under the influence of other objects.

Chapter 5

Implementation

This section deals with the practicalities of implementing the requirements laid out for the project. The primary areas chosen for discussion are the choice of the language, how the system detects and reacts to a collision between two objects, and the method the system uses to relate objects in a DIG.

Language Choice

The MAVERIK toolset is a middle layer between the upper layer of the programming language (C) and the lower layer of OpenGL (in the case of this system –other 3D accelerator languages are supported, for example IrisGL for Silicon Graphics workstations, and Glide for older Voodoo-based graphics cards). It meant that all that was required to begin work on this project was an understanding of the C programming language, and a knowledge of how to implement the mathematical solutions. Problems such as how to represent a set of three dimensional objects in an efficient manner are handled by the MAVERIK library, and a simple user interface is provided to the C programmer. This library has made it possible for the project to avoid the issue of how to represent 3D-space, and while this is very prevalent for VE designers, it is outside the focus of this project.

The Java 3D (J3D) API is a library similar to MAVERIK, except that it uses Java rather than C as it's high-level language. Therefore, this project could have feasibly been implemented in a purely Object-Oriented context, providing more manageable code re-usability and a more organized abstraction. However;

- Interpretive languages are not reknowned for their speed of execution. A C++ implementation (if executed correctly) could be several orders of magnitude faster than an equally competent Java implementation.
- A set of C++ header files with an efficient interface is just as re-usable as a Java class with the same.

It was therefore decided that a C++/MAVERIK implementation would be able to fulfill the speed requirement more effectively than a Java/J3D one.

Physical Information

MAVERIK data structures hold a variety of information to define the 3D objects they represent. This information is not always common across datatypes; for example `MAV_sphere` holds a parameter radius, but this would be meaningless if applied to `MAV_box`, which instead holds variables for the size in the X, Y and Z dimensions. However, all objects hold a 4x4 transformation matrix that contains the position of the local centre of the object (which is also the centre of mass for the MAVERIK-defined primitives), the orientation parameters and the scale size.

In order to hold the information necessary to create an accurate physical representation of objects, an additional data structure with the working name of `MAV_newton` (because the mechanics are Newtonian) was created. This holds both state information (three position vectors and three rotation vectors) and physical attributes of the object (coefficients of friction and restitution, inertial tensor). It also contains information that is useful for the program to know (the ID and time of the last message received, whether or not the object is at rest relative to the floor, whether there is enough information to use Verlet integration or if Euler should be used instead). Finally, it contains a pointer to the MAVERIK datatype mentioned above, so a `MAV_newton` giving information about a `MAV_box` also contains the information MAVERIK needs about the box encapsulated within it.

As the MAVERIK system itself was not written to take this state information into account, it is still the object in the data structure that MAVERIK uses in order to draw it correctly. To this end, once the information is calculated it must be sent to the transformation matrix inside the primitive's object in order for the position to be correctly rendered.

The contents of the `MAV_newton` type was derived mainly from empirical evidence (it is obvious that the object's position and basic information should be known), but was also dependent on the equations used to calculate the collision response (the inertia tensor [7] in particular was solely included because it is used in calculating the angular momentum).

NB: for completeness, the structure of the `MAV_newton` datatype is included in Appendix B.

Collision Detection

As mentioned in the Literature Review, MAVERIK contains built-in methods for calculating bounding volumes of objects and testing those volumes for intersects with objects. In order for the system to detect a collision, it has to iterate over the array containing the `MAV_newton` pointers, create a bounding box for each item and check that volume against all objects in an SMS in order to calculate a collision. This method returns another SMS that

contains the objects which intersect the object being tested. As a side effect, this SMS also contains the object being tested, so checks were implemented to cirumvent this false positive.

The bounding boxes used by MAVERIK are axis-aligned cubes, so the volume they specify can be larger than the actual space occupied (for example, a sphere, or a cuboid with an orientation other than [0,0,0]). As this is the case, methods had to be used to ascertain whether or not the shortest distance between the objects was less than zero (constituting an overlap, therefore a collision). Two callbacks are implemented to aid in this task – `mav_callbackGetSurfaceDist` and `mav_callbackGetImpact` – which get the distance from the centre of the object to the point of impact, and the local co-ordinates of the point of impact, respectively.

Once a collision has been confirmed as being between two objects and not just their bounding boxes, a message object is created and added to the end of the message queue. However, a check is performed to ensure that the message isn't just the reverse of the previous message added to the queue – if two objects collide, then two collisions will be generated – one of object A hitting B, and one of B hitting A. When this check is complete, the message queue in each frame will contain a single message relating to every collision that occurred in that frame. The message queue can then be read and the collision responses calculated.

NB: The message processing function and message object layout used is adapted from [5]. The packet object specification is in Appendix B

Collision Response

Once the collision detection loop has terminated successfully, the message queue is parsed for collision messages. The first check is to get the resultant velocity of the two objects (velocity of A subtract the velocity of B), and then calculate the dot product of that vector with the normal to the collision (pointing towards A). If this dot product is negative then the objects are moving towards each other, and a collision has occurred. If the result is zero, then the objects are moving in parallel, and if it is positive then they are moving away from each other. The final case can occur if the bounding boxes are still overlapping after a collision has been performed.

If the result is negative, then equation (6) is computed in order to give a scalar value that must be applied to the collision normal in order to give the impulsive force that the collision produces. This force is divided by the objects' mass (in turn) to give the acceleration, and this is applied to the objects in order to get their velocity as they leave the collision, and from that their next position.

For rotational information, the scalar is applied to the vector perpendicular to the velocity of the object before the collision and the normal to the collision. This gives a value proportional to the incoming velocity (the normal has a magnitude of one) in the axes in which rotation should be performed. After the rotation matrix has been calculated, it is re-orthagonalised in order to reduce the possibility of rounding errors which would otherwise increase proportionally to the running time of the system [4].

Dynamic Interaction Graphs

In order to reduce operational overheads as much as possible, the DIG system is implemented as a two-dimensional array (limited in both directions by a global constant that defined the maximum number of objects it is possible to create – this means all of the objects can be in a DIG of their own, or all joined together in one DIG). Each object is initially created in an empty DIG, and checks are made on the collision response scalars to see if the object only has a miniscule force pushing it away from the collision. If the force is negligible, then the objects position vectors are set to the rest position, it's velocity (perpendicular to the collision) and resultant force set to zero, and the DIGs altered.

The DIGs are altered using a simple array search that gets the DIG number, the index of the object in that DIG and the total number of objects in the DIG of both the objects in question (the object whose response scalars were lower than the criteria is termed the source, and the object it collided with is termed the target). This information is then used to remove the source object from its DIG, re-initialize that DIG (re-order it if the source object was not the only object in there or not at the end of the list), and add it to the end of the target graph.

[5] used an idea of a cog list, a pointer to an array of cogs that were directly influenced by the cog in question (neighbours in the primary compass directions). This approach was deemed unsuitable for this application, because a number of checks had to be implemented in order to ensure that the same cog was not receiving the rotate message packet from two independent sources. The method used in this project means that an operation performed on one object in a DIG just needs to be iterated over every object in that array index, with little-to-no overhead in checking mechanisms.

Chapter 6

Testing

Overview

The comprehensive testing of something as non-deterministic as a physics engine is always going to be a very intensive and time-consuming task, depending on the accuracy required. The simulation can be run, and the effects watched by a human observer, and if the objects appear to be interacting in the correct way, then the simulation has succeeded in one respect. However, a human observer will not be able to tell that all velocities were out by a factor of ten, for example – if the system seems consistent, then it will appear correct.

Conversely, a human observer can also tell better then a computer observer that an object collision is modelled incorrectly – if parts of the objects are passing through each other, or if the object is sliding instead of rolling.

As absolute physical accuracy was not a requirement, the following tests are judged by a human observer with no measurements. A debug mode is available to advance the simulation by one frame at a time if there is ever any doubt as to a simulation's correctness.

The testing was conducted through the creation of scenarios (as documented in [16 section 20.3.2]) that would demonstrate to the user whether or not the simulation had the functionality that was expected of it.

Scenarios

The scenarios will be divided into two groups in order to test the two fundamental requirements – physical behaviour and DIG creation. The following scenarios all take place in a six-sided room, where the boundaries do not have a physical wrapper. They are suspended in the air, but are used as triggers for collision detection (the collision detection methods register a collision with the type of the boundary and handle it specially). The results of these scenarios can be found in Appendix C.

Physical Behaviour

- Create an object such that its lowest extremity is touching the lower plane. This object should be created at rest, and then have its position bound by the constraints imposed on objects.

- Create an object vertically displaced from the lower plane, so it has an initial velocity. This object would then bounce off the lower plane, having its velocity reduced by a factor of its coefficient of restitution, until it has low enough escape velocity for the constraints to register that it is (effectively) at rest relative to the floor. At this point the result should be identical to the first scenario.

- Two scenarios must be created that involve the registering of collisions between two objects (non-boundary type) when one is moving and the other is a) moving, and b) at rest on the lower plane.

- The above scenario may involve the collision of objects with the side planes – these collisions should be examined when the object is both moving and at rest relative to the lower plane.

Dynamic Interaction Graphs

- Subject to the physical constraints of the system, two objects must be created that will eventually be at rest relative to each other, and therefore should be in the same DIG. Once the operation is completed, the code status must be examined to ensure that the objects are indeed in the same DIG.

- Once there are two objects in a DIG (see above scenario), then another object that is not in the same graph must be created in such a manner that it will disturb the objects in the DIG. The effects of which should be that when the collision is registered, a collision message is sent to each object in the DIG (even if only one was involved in the collision), then the objects in the same graph should be allocated their own graphs, and the collisions resolved.

- For the issues of scalability, these tests should be repeated when there are more than two objects in a DIG, just to ensure that the messages are passed correctly, and that the collisions are resolved correctly, all within a reasonable frame rate.

Chapter 7

Conclusion

Limitations

Several limitations are imposed on the simulation that were not envisaged in the requirements stage.

Firstly, only spheres are modelled correctly. This is due to an inability to correctly calculate the point of intersect on a box whose orientation is non-standard (i.e. the constraints of the bounding box are not the constraints of the box itself). Also, the addition of a system of moments to control the balancing of a box on an edge was deemed to be a misallocation of resources – it has no relevance on the main subject matter of the project, and implementing it would be comparable to negligence. The fulfillment of the primary requirements could suitably be judged only through the use of spheres.

However, the DIG functionality is also quite limited. If there are more than two objects that should be assigned to the same graph, then the mechanisms for checking this eventuality break down. This is due to the operations not being able to apply a collision to an object in a graph without breaking the graph up into its components – this way, an object can not join a graph that is already formed and has more than one component object.

Problems Encountered

Programming

Several of the MAVERIK built-in methods had a safety restriction that meant no division would be performed if the denominator was less than 0.0000001 (1e-7). This seems to be a redundant, or at least overly-secure, check, seeing as the precision of floating point numbers is valid up to 1e-44 (http://babbage.cs.qc.edu/courses/cs341/IEEE-754.html), and it generally caused more errors than would otherwise have been present. To this end, the functions involved had to be copied from the MAVERIK source code and re-written in the system's own header files without the constraint.

Conceptual

The DIG functionality was not scalable – it could not perform the correct operations when a graph should contain three objects. It is believed that the fault lies in the message dispatch code, such that it cannot differentiate between a collision that should dislodge objects and put them in unique DIGs, and a collision that is of sufficiently small velocity to mean that the object should be added to the DIG in question. This resulted in the error outlined in the DIG scenario test case in Appendix C.

Although the functionality for the DIG concept had several limitations that may deem the project to be incomplete with respect to the requirements, measurements of the frame rate indicate that high frame rates (~40fps) were obtained even when a reasonable number of objects were being processed. This should instill faith in this approach to physically-based modelling – operation speeds could easily be increased by writing the graphical output directly in GL (or an equivalent) rather than using a middle-layer language, and also by using a polygonial-based display method rather than constructive solid geometry – this would lead to possibilities such as back-face culling, meaning the graphics hardware is not displaying polygons that cannot be seen.

Concluding Remarks

This project set out to show that physical realism in games/Virtual Environments could feasibly be created using the grassroots approach of implementing Newtonian mechanics equations and the fundamentals of rigid body dynamics. Examples of how these equations have previously been adapted for a low-accuracy, high-speed environment were provided, and reasons for this project's existance were outlined. The term 'believability' was introduced, explaining that ultimate physical realism is not necessary, but if the behaviour looks suitable, then the user will be suitably fooled into thinking the environment is real. The idea of a consistent and coherant physical environment was put forward, with each object present exterting some physical behaviour on any others it may interact with.

The collision detection complexity problem was outlined, with special detail given to the problem of objects that are known to be influenced by other objects. A method for limiting the complexity of the collision detection operation was outlined, and the concept of the Dynamic Interaction Graph was introduced, with considerable background material.

The ways in which a physically-realistic system that incorporated DIG concepts and functionality would be computationally more efficient than one that did not were explained. An implementation of a DIG and physical realism-based system was provided, although its concept of DIG was non-scalable – this is a failure with respect to the requirements.

However, the system still demonstrated that the DIG concept could be applied to a physically-based system with some success. If more resources were available, then there is confidence that a fully-scalable solution could be produced.

The test programs and source code are available from:
(1) http://www.kfj.f2s.com/rich/project/
(2) http://www.bath.ac.uk/~ma0rpj/project/

NB – the second URL will only be valid until July 2004.

Bibliography

Physics

[1] Jamie Cheng: Creating a Fast and Stable 2D Pixel Perfect Rigid-Body Simulation using Verlet Integration:
http://jamie.illusiondesign.net/PPCD/

[2]: Thomas Jakobsen: Advanced Character Physics:
http://www.ioi.dk/Homepages/thomasj/publications/gdc2001.htm

[3]: Bond Graphs website:
http://www.bondgraphs.com

[4]: Chris Hecker's papers on collision response (2D and 3D)
http://www.d6.com/users/checker

[5]: Prof. Phil Willis' MAVERIK example program (Gearwheels)
Included on CD

[6]: Coloumb friction model
http://encyclopedia.thefreedictionary.com/Frictional%20coefficient

[7]: Inertial Tensor calculations:
http://scienceworld.wolfram.com/physics/MomentofInertia.html

Rigid Body equations were taken from Halfman, 1962 – *Dynamics – Particles, Rigid Bodies And Systems, Vol. 1*, Addison-Wesley

Contemporary Engines

[8]: Framework for physically-based modelling in Virtual Environments
http://aig.cs.man.ac.uk/gallery/iota/iota.html

[9]: Prof. Phil Willis : VIPER Summary:
Appendix A

[10]: Truck Dismount
http://jet.ro/dismount/

[11]: Bridge Builder
http://www.chroniclogic.com/

[12]: HAVOK SDK
http://www.havok.com/

[13]: Deus Ex 2
http://www.deusex.com/

[14]: Half Life 2
http://www.sierra.com/product.do?gamePlatformId=470

[15]: Half Life 2 demo engine room:
http://pc.ign.com/articles/400/400985p1.html
(paragraph 12)

Software Process

[16]: Ian Sommerville - Software Engineering, Sixth Edition, ISBN 0-201-39815-X

[17]: Synopsis provided by Professor Willis as basis for this project
Appendix A

[18]: IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specifications

Referenced throughout the document creation process:

Project Dissertation Notes, Alwyn Barry,
http://vlc.bath.ac.uk/courses/1/CM30082/content/_8593_1/ProjectDissertation.pdf

Sample First-Class Software Development Dissertation, Andrew Monaghan,
http://vlc.bath.ac.uk/courses/1/CM30082/content/_9360_1/AndrewMonaghan2003.pdf

Appendices

Appendix A: Requirements Data

Project Synopsis

Title: Physics for Games/Virtual Reality

Description:

Most computer games showing 3D worlds have either poor physics or a very limited range of physics. Often this is implemented by specialised code "attached" to those objects which erspond. As a result, most of the VR world does not behave like the real world at all and it little more than 3D wallpaper against which the action takes place. As part of our own research, we are developing a simple mechanism which supports interaction between objects, with possibilities for mechanical, electrical or fluid effects. The aim of this project is to take our basic model and develop it to show a small world in which all objects behave as expected, using these aspects of physics.

Pre-requisite knowledge:

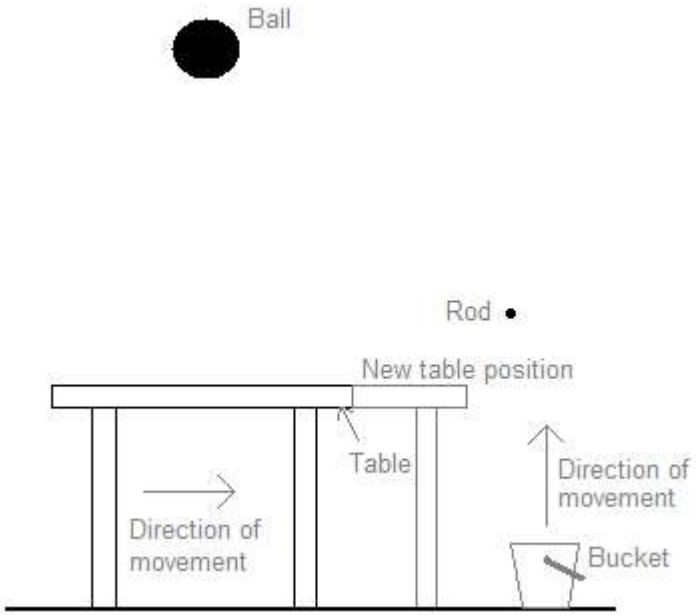
Needs programming skills in C, C++ or similar. Could be Unix or Windows based.

Source Material:

Demonstrator programme (in Visual C++ but you can also do the project in C or C++ in a Unix environment if you prefer). Documents describing our own research plans.

Example Scenario

The following is an example scenario that was originally included with the Literature Survey. It has since been moved to this appendix, because it is felt to be more suitable as an example of a requirements-harnessing scenario than an accurate representation of what the simulation is currently capable of.



Initial DIGs:
Ball
Table
Bucket, handle
Rod

i) A spherical ball is created above a wooden table. It is immediately subject to gravitational acceleration. The next position after a time t is calculated using Euler equations, and from then on the progression is mapped using Verlet equations.
ii) At some stage, one position of the ball's lowest point is above the table's top surface, and the next is below it. This is a trivial example seeing as the ball is only moving in one dimension. The point of intersection is found by continually bisecting along the vector between the two points.
iii) The velocity at this point can be calculated using Euler's equations. By reversing the vector in the direction of the acceleration, the velocity after the collision is found (if the collision isn't perfectly elastic, then the velocity should be multiplied by a dampening factor).
iv) Euler is again used to get the correct position of the ball at the same timestep as that when the ball clashed with the table surface, and one more time to get the next position after that. Then Verlet can be used again.
v) After a series of bounces, the ball will eventually come to rest (assuming the table isn't perfectly elastic). At this point, the ball will be added to the DIG of the table*, and any movements that affect one could potentially effect the other.

- For the ball to remain at rest, there will have to be a resultant force of zero – i.e. a reaction against gravity must act from the table surface on the ball. Current implementation is unknown, but it is assumed that it will be part of the DIG's consequences.
  Updated DIG:
  Ball added to table

vi) The table is then slid along the floor for a short period of time such that the frictional coefficient between the table and ball cause the ball to remain still.
i.e. – a small (lower than the static frictional force between the ball and table) constant force is applied to the table.

vii) The table is then stopped suddenly, causing the ball to roll along the table.
i.e. – a large (greater than the static frictional force) force is applied to the table in the direction opposing the motion. Half of this force will be applied on the contact point of the ball and table, and the top of the ball in the opposite direction. This will cause the ball to roll towards the end of the table (Velocities will be calculated in a similar way to i).

viii) The ball rolls into a fortuitously placed bucket.
Updated DIG:
Ball removed from table, added to bucket, handle.

ix) A constant upwards force is applied to the handle of the bucket, which tends to the vertical. When the handle is vertical, the bucket begins to move up.
- The handle pivots around the attachment to the bucket chassis with angular velocity calculated from the force applied.

x) The bucket and ball continue to accelerate upwards until the edge of the bucket hits the rod. At this point, the rod is added to the DIG of handle/bucket/ball, and the bucket rotates and slides anticlockwise around the rod.
Updated DIG:
Rod added to ball, bucket, handle

xi) When the bucket rotates past a certain angle, the ball will begin to roll out
- The angle needed will be a result of the frictional forces between the bucket and ball (presumably less than that of the table and ball, due to the bucket being metal)

xii) The ball will exit the bucket and begin to fall towards the table again. The bucket will swing back around the handle after it is free of the rod, and will continue to oscillate until the friction between the handle and container dampen it to zero. The ball will bounce along the table in a similar manner to before, but this time it will have horizontal momentum and rotational velocity. Eventually it will stop bouncing and rolling, and the system will be at rest.
Final DIGs:
Bucket, Handle,
Ball, table (assuming the ball didn't have enough momentum to roll off the edge of the table).
Rod

* It is unknown at this time whether or not the ball should be added and removed from the DIG for each contact point. Assumption: If the table is not moving, then the DIG will only be updated when the ball comes to rest. However, if the ball bounces on a surface in motion, then each momentary contact will influence the ball's resultant trajectory, so it will need to be added, calculations performed, and removed. This will introduce another overhead. It is possible that the calculations could be performed without adding the object to the DIG, but the theory behind the implementation only allows objects in the same DIG to have an effect on each other.

VIPER Summary

Notes on this document: Any references are self-contained, and not included in the project's overall reference section. This document was provided by Prof. Phil Willis as a part of the initial documentation given to project students in order to get an idea of what is required. The document has been formatted to fit in with the style of the rest of the document, but is otherwise provided as-is.

We are familiar with the real world and know how to use it. For this reason we can manipulate unfamiliar objects and use objects in novel ways. When we construct something new, we do so because we are predicting properties which we want the new object to have. Those properties emerge as a consequence of the construction. Engineering is based on this premise but so is a broad range of our everyday experience.

We propose to investigate the systems architecture issues to permit users to bring objects together anywhere in a large VE, to allow almost all objects to affect each other, to permit the emergence of new behaviour, and to support in a unified way a range of physical properties (mechanical, electrical, hydraulic etc). The application area will be a large distributed VE with thousands of objects and potentially multiple distributed users. We do not propose a scientifically accurate simulation, for example by producing full dynamics solutions. Our immediate application area is entertainment games but we are proposing a generality well beyond current game technology, which we believe will then be applicable in many VE training applications. It is an objective of this research that we will demonstrate a VE which behaves sufficiently like the real world that users will be confident both about manipulating it and the consequent outcomes.

We propose to build a software system which supports the emergence of physical behaviour in large scale virtual environments, to include mechanics, electricity and hydraulics as examples. Users of such environments will be able to manipulate objects much as they would in the real world. Objects will stay on a desk because of the physics, not because of an arbitrary transform which locates them there. Infrastructure such as lighting in a building will operate as a consequence of the wiring, the switches and the power supply, not as a consequence of specialised code. Water will flow because of the piping, not because of code animating a tap.

This system will therefore support worlds in which the behaviour emerges because the physics is modelled alongside the geometry. As a result, consequences unexpected by the designer of the world will be possible, whether in creative use of existing objects or in using simple objects to make more complex constructions, such as mechanisms. Furthermore, we will be able to use mechanisms as the engines of animated objects (as existing systems do in the abstract) yet still have them interact fully with other objects and be extendable by the user of the VE. This requires a profound rethink of the way we model and render the 3D world, as well as how we incorporate behaviour in this general way.

We summarise the major research issues as follows.

- Representing all common physical interactions between objects
- Giving all objects properties and predictable behaviours
- Unifying the representation of the basic physics of virtual environments
- Applying constraints in a dynamically-changing world
- Maintaining model consistency in a dynamically-changing world
- Rendering a dynamically-changing world

The Real Environment

Consider the following scenario. A man walks along a busy street, carrying a small flat-packed self-assembly table. The on-coming pedestrians steer their way around him. He crosses the road, weaving carefully through the traffic and enters a residential building. Summoning the lift he rides to the floor where he lives and opens the door of his flat with a key. He switches on the light but the bulb blows and he is unable to reach it to replace it. He decides there is enough daylight to assemble the table, which he does. He then drags the table below the light fitting, takes a spare bulb from a sideboard drawer, climbs on the table and exchanges the bulbs. The old bulb is still warm but not uncomfortably so. Stepping down he tries the light switch once more, with no result. He realises the failing bulb must have blown the fuse and goes to the fuse box in the kitchen and changes the fuse. Returning to the living room he tries the switch once again and is rewarded with light. For the first time he notices that the mechanical clock on the sideboard is running slow, so he moves the minute hand around until both hands show the correct time. He moves his portable television from the floor to the table. There is also space for the lava lamp he bought yesterday, so that also goes on the table. He drags the table to the corner of the room and plugs in both television and lamp. He switches on the lamp, the kind with multiple pieces of coloured foils, and switches it on. After a few minutes of warming up, the foil pieces gradually convect. Realising his hands are dirty, he returns to the kitchen, puts a plug in the sink and runs enough warm water to wash. He pulls out the plug and the water gradually drains away.

This scenario of mundane happenings exhibits a number of features that are routinely found in reality but are broadly lacking in virtual environments. All any of them could be fabricated by writing the appropriate dedicated code but we are interested in making the everyday physics behind these events work properly in the virtual world. The reason for this is that the real world does not consist of fixed scenarios that we have to “solve”, as is the case in most exploration games for example. Rather we know how the real world works. A VE which works the same way is easier to use, adaptable to our needs and richer in possibilities. When our subject moved the table under the light fitting in order to reach the bulb, he was using the table in a way that its designer did not think of. This is an example of emergent behaviour, behaviour which emerges as a consequence of the properties of the object being used in an unplanned way, not as a consequence of some built-in code determining a specific use. Doubtless the clock can be updated by dedicated code but it is the physical linking of the two hands, in a 12:1 ratio, which permits the user to wind one hand and get the clock to show the

correct time. The temperature of a heated object depends on its own properties – specific heat – but is driven by the external application of heat and the laws of cooling and convection. Whether a light comes on depends on the existence of a complete circuit, from mains to fuse to switch to bulb and back again. It cannot exhibit its full rich behaviour if hard-coded as "if switch then light". The behaviour of a crowd of people and a flow of cars can be modelled with flocking but they are not immune to other influences, whether predictable (traffic lights) or not (someone crossing unexpectedly). The rules for flat pack assembly are simple physical constraints. Catching them correctly not only ensures any other flat pack item can be assembled but it also permits disassembly. Similar constraints apply to the items resting on a table, water staying in a sink, the way a drawer opens and the way a lift moves in a shaft. A hinge both keeps the door attached to the frame and also causes it to rotate.

These simple mechanisms are responsible for making the world mutable; that is, we can manipulate objects, reposition them and use them in ways determined by us. Our everyday experience depends on this. If we take an object from a table, the table remains where it is. If we drag the table, the objects on it move with it. If we tilt the table, some objects may slide off. These effects are not hierarchical – an object which slides off a table is initially affected by the tilt but ultimately separates from the table. Rather they are a manifestation of the physics of the world. The appearance of an object is a consequence of a physical interaction between the object's properties (e.g. surface texture) and the ambient lighting. Similarly, emergent behaviour should arise from the physics "out there" working with the object's properties, such as mass. It should not be built in to the current objects or their current disposition. All object properties and the physics behind behaviour therefore need to be modelled independently in a generic virtual environment. Finally we note that in computer games, it is often desirable for the physical laws to change dynamically. If our subject climbs through his full-length mirror into a world where magic is possible, then ordinary physics should be replaced or enhanced by the physics of magic. Current commercial systems, especially in gaming, emphasise appearance, offer pre-canned behaviours, and often reduce to finding the one solution built in by the creator. In the rare examples where this is not always so (Doom3 and HalfLife2 for example), distributed behaviour is not supported.

The Virtual Environment

In what follows, we will use the term designer for the person who builds the VE. We will use the term user for the person exploring the VE.

The high-level Inventor product has some of the characteristics we need for implementation but not enough. It takes a strongly hierarchical view which we don't want and this is hard to sidestep. The two low-level standards OpenGL and VRML have a common heritage. Scene graphs evolved as a way of representing a 3D model in a way that was efficient for the rendering pipeline of specialised graphics computers. They include nodes to represent the scene geometry, the lighting and the code to generate animations. They are well-suited to displaying a scene and allowing a user to move around it. They are however poor at representing correct relationships between the components of a scene. This case is well-

argued by Lescinsky et al. [LES02] who point out that even a simple relationship, such as "X is on top of Y", requires an artificial hierarchy to be imposed. This hierarchy suits the rendering process but is not true to the modelling needs: X is not part of Y. This makes full interaction with the scene difficult. Users cannot in general manipulate objects unless the designer of the VE anticipated this and provided the necessary code.

VRML objects will not interact with each other as they do in the real world. VRML provides statically defined routes to pass messages between objects. The designer must decide which objects can interact and hardwire the routes needed. It is not feasible to permit any object to affect any other and indeed it is tedious to allow even one object to affect many. As a result, behaviour arising from arbitrary physical interaction imposed by the user but unanticipated by the designer is impossible.

Physical modelling of dynamics provides an answer to this. For example, it is then possible for the user to make arbitrary virtual assemblies and for these assemblies to operate as expected. A thorough dynamic modelling is however a non-trivial computational problem, with possible detrimental effects on the real-time VE. We believe that this is not necessary in order to build a plausible world which still has emergent behaviour. The everyday physics familiar to us when we pick up, move or stack objects involves little computation. Frictional forces and rigid collisions are similarly undemanding. Elastic interactions are much harder but the need for fully correct elastic calculations is rare in the everyday way in which we manipulate the objects in our world. Complex dynamics are simplified when only rigid objects are involved. This means we need a consistent representation for the physical forces in the world. Bond graphs are one such representation.

Bond graphs

A Bond graph is a schematic of a system, representing power exchange. The links connecting the nodes show the flow of power. They represent pairs of properties, one across and one along a link (voltage/current for example: hence power). The nodes in the graph correspond to physical constraints. Such a graph closely maps the structure of the physical system, such as the arrangement of junctions and components in an electrical circuit. Importantly, it can be constructed from the arrangement of the system, before any decisions about causality need to be made. This is important to us because we need causality to emerge from the arrangement of each system.

Bond graphs have been studied for 50 years and are thus a mature form. The technique has been applied to thermodynamic, hydraulic, chemical, mechanical, electrical and electronic systems and there is software support available. Bond graphs are sophisticated tools and the area is still the subject of active research. We do not propose to make a contribution in this

area. Rather we will adapt the experience of Bond graphs to our specific needs, leaving open the possibility of using it to model more sophisticated systems later in the project.

In addition, our approach is not one of modelling "a (large) system"; it is one of modelling thousands of (small) systems which vary all the time in their configuration. This suggests we should think of an extensive and dynamic cluster of Bond-like graphs with the nodes being objects and the links being the contact between them, transmitting the physics. Exploring how to do this well and calculating the consequent physical results is at the heart of our proposed investigation.

Proposed Approach

Dynamic interaction graphs

If the user is free to manipulate any object, taking it to new places in the VE and making assemblies, then a key issue is knowing which objects can affect which others at any given time. We need a fluid way of determining this and propose to use graph structures. Those objects in contact are dynamically linked together in such a structure. In any given VE, there will be many graphs, each perhaps representing an assembly of objects (such as the items on a desk) or a mechanism (such as a mechanical clock). The graphs are not fixed however. As the user manipulates the objects in the VE, graphs come into being, grow, merge, split, shrink and disappear in response.

These graphs do more than record which objects may affect which others. They also provide the means to resolve the consequences. The links of the graph show how to propagate event messages between the objects, so that actions can be coordinated and their effect determined. For this reason we will use the term Dynamic Interaction Graphs, or DIGs for short. Multiple DIGs thus become the key items that have to be managed to facilitate the physical interaction of the associated objects.

The objects in a scene are thus not linked in a scene graph. Each object instance must know where it is located and this transform must be actively maintained. An object which cannot be broken down by the user (an atomic object) can be modelled in any way that can be rendered, including by a hierarchical structure. What is key is that the dynamic aggregation of objects must be based on an easy-to-maintain approach. DIGs achieve this.

Messages

Events affecting one object may have a consequence on another object. The links of a DIG correspond to pairwise contact between objects. So, when one object is moved (say), it can send a message to its graph neighbours so that the consequences on them can be resolved. This also applies to their neighbours in turn. VRML timestamps event messages to ensure

that a given node does not action a message twice. This also prevents indefinite circulation when the graph contains circuits. We will use the same mechanism. We have tested an implementation based on a single message queue separate from the graph itself. This works well and has the advantage of providing easy monitoring and testing, as well as the later possibility of the event queue being on a different processor to the DIG.

Messages have to interact with the physics, such as friction and mass. We will provide a library to support this. As we envisage thousands of DIGs, with 10-100 objects each, this will be written in re-entrant code. In this way the local state will be in the object and the physics itself will be "out there". As we extend the range of physical processes to include electricity, hydraulics etc, so we will provide new library procedures. As described earlier, we will not be attempting to simulate complex dynamics etc: the aim is to demonstrate credible system solutions for everyday interaction with the VE, such as within an advanced gaming environment.

Bringing emergent behaviour to virtual environments

We propose to use DIGs as the fundamental data structure representing the VE. Every object will belong to a DIG (even if it is the only member). When two or more objects are brought close enough together to interact, they will be linked into a DIG so that the physics can be resolved. If they later separate, the link is broken and so they no longer interact. For example, if a book is placed on a desk, the book and desk will temporarily become part of the same DIG. The associated messaging system permits the statics to be resolved: the book sits on the desk but does not fall through it. It permits simple dynamics to be resolved: the book can be slid across the desk and friction will bring it to a halt.

DIGs will also support the hierarchical function of a scene graph – if we move the desk, the book goes with the desk – but without having the structural hierarchy of a scene graph. The latter is evident when the desk is tilted: the DIG allows the book to slide off, the scene graph does not. A DIG shows hierarchical behaviour only where the physics requires it, while the scene graph rather imposes this through geometry. The DIG readily allows us to move the book elsewhere, where it will join another DIG.

Each instance of an object therefore has to know the absolute transform that locates it in the VE. These transforms must be maintained when objects are moved, especially when a group of them are moved. Lesinsky et al.[1] show how this dependency can be achieved without a strict scene graph. We will need to extend their approach to permit moving aggregations of objects rather than objects linked with the "on top of" operator that they use.

[1] Gordon Lesinsky, Costa Tourna, Alex Goldin, Max Fudim and Amit Cohen, "Interactive Scene Manipulation in the Virtue3D System", Proc. Web3D '02 Conference, ACM, pp. 127—135, 2002.

We can also use the links of the DIG to record physical information such as resistance. This is in keeping with Bond graphs (VRML has no concept of value associated with links). We can use this information in creative ways within a VE. For example, exploratory messages have a rigorous application. A user might bolt a door. If another user then tries to open the door, a message can be sent around the DIG to check for freedom of movement. Only if a positive answer is received will a message making the movement be launched. The importance of this cannot be overstated: we envisage a world where users can reconstruct freely and we must have a general method to cope with this. The door might have been held closed by a heavy object, rather than with the in-built bolt. The same argument applies generally, whether to motors, electrical circuits, water pressure in a pipe etc. Some of these constructions will produce viable results, others will be contradictory and an exploratory message allows this to be checked.

We will also need a corresponding constraints system associated with the objects. In the overall spirit of demonstrating large-scale generic solutions we will confine this to a simple range of constraints, permitting us to concentrate on managing the generation of behaviours. Only later, once the messaging system is well understood, will we need to explore constraints more deeply.

So far we have largely concentrated on a user moving and assembling objects. DIGs also provide a simple way of encapsulating free-standing mechanisms. A useful example of this is flocking behaviour, used to simulate crowds and traffic in a busy city. Here "contact" is defined in a more elastic way, over a limited range. A link which goes out of range is broken and so flocks can split (or merge) according to external influences (traffic lights or a narrow path for examples).

Overall this approach is one of geometry plus physics plus constraints plus mediation via messages. This approach reduces the programming needed by the designer, partly because actions can be captured but mainly because the behaviour emerges.

Separate modelling of behaviour and geometry

Some aspects of a VE are best thought of as geometry-only (i.e. with no generic behaviour). This includes the ground and the floors and walls of buildings. These objects do in fact have a limited behaviour – solidity – but this is the very property that, for most purposes limits the behaviour of other objects, rather than communicates it. In other words, these objects bound the extent of our DIGs. A book on one desk cannot affect a book on another desk, even though both desks are on the same floor. Of course this does not mean that a floor cannot have behaviour, as it would if it was destroyable (which would affect all desks, books etc on that floor). It is a practical observation that geometry-only objects have an important role, which is made explicit in our approach.

Similarly we need behaviour-only objects. For example, the electrical wiring of an office building is not seen, so there is little point in modelling its geometry. We can however model the behaviour, with a DIG which connects all relevant switches and sockets, such that turning a switch off turns off the connected light (etc). The heating (or cooling) of a room is invisible, even in reality, but the physical consequences are real enough.

Commonly we will require geometry and behaviour. The rings of an electric cooker transfer heat by contact yet also need geometric models to be seen. The appearance of the ring varies with temperature. DIGs can cope both with changing the appearance and the transfer of heat by contact.

Instancing of geometry is important in constructing large VEs. Our approach extends this to instancing behaviour. Suppose we construct a DIG which constrains a drawer to slide. Having done that once, we can build a variety of drawer units, desks etc using this one behaviour. Similarly a hinge DIG can be used for a door, a gate or a window. We are free to combine various geometries with various behaviours, giving a richer VE. In fact, we have the same "freedom of assembly" that we have in the real world, whether for flat pack furniture or less-structured objects.

Appendix B: Type Definitions

This appendix holds type definitions that are useful to have to hand when reading the document, if none of the other code is available.

MAV_Newton members

`int objectType;`
: Uses defines to cast to the correct type at draw time (SPHERE, BOX or RECTANGLE)

`int m_id;`
: Identifier of most recent message received by this cog

`int db_id;`
: Debug ID, used to reference object – also the object's position in the global array storing all `MAV_newtons`

`int mess_type;`
: The type of the last message received

`int dig_id;`
: The ID of the DIG this object belongs to

`double mess_time;`
: The time of the last message received

`bool floor;`
: If true, ball is rolling/sliding along the floor and not bouncing (switch from impulse-based to constraint-based modelling)

`bool collided;`
: Used to see if the collision has been processed already

`bool euler;`
: Used to see if euler integration should be used to get the next position

`MAV_vector currPos, prevPos, nextPos;`
: The current, previous and next positions of the centre of mass of this object

`MAV_vector currRot, prevRot, nextRot;`
: The current, previous and next rotations of the object

`MAV_matrix iTensor, iti, itiws;`
: The object's inertia tensor in local space, the inertial tensor in world space, and the inverse of the world-space tensor.

`float moInertia;`
: Moment of inertia

`MAV_vector resForce, torque;`
: Resultant linear and angular forces on the centre of mass.

`MAV_vector cMass;`
: Centre of mass

`MAV_vector angMom;`
: Angular momentum

`MAV_vector velocity, omega;`
: Linear/angular velocity

`float mass;`
: Mass of object (acting at centre of mass)

`float rest;`
: Coefficient of restitution

`struct s_newton *origin;`
Pointer to the sender of the last message this object received
`MAV_object *object;`
MAVERIK's object data
`MAV_surfaceParams *sp;`
The surface parameters of the object, defining how it should be rendered
`MAV_matrix matrix;`
The transformation matrix of the object

Packet_object definition

`MAV_newton *me;`
Pointer to the transmitting cog.
`MAV_newton *you;`
Pointer to destination cog.
`int m_id;`
Unique identifier for this message.
`int db_id;`
Transmitting cog identifier.
`int mess_type;`
The type of the message (at the moment only COLLISION is implemented, but this allows for scalability.
`double mess_time;`
The time of the message creation.
`MAV_vector direction;`
Vector defining the direction between the centres of the objects involved with the collision (NULL if colliding with a plane, as the centre of the plane has no relevance to a collision).
`packet_object *next;`
Next packet in the list – always set to NULL on packet creation, but altered by the message queue functions in order to create a linked list of messages.

Appendix C: Test Scenarios

Physical Behaviour

- All of the physical behaviour scenarios were completed successfully.
- There were, however, some errors present in non-test case scenarios, such that objects would be given an invalid position after their DIG allocation had gone through a certain procedure. This is being considered an error with the DIG functionality, and not the physical behaviour. This can result in the disappearance of objects in the test applications provided, but it is not an error in the physical modelling.

Dynamic Interaction Graphs

- Subject to the physical constraints of the system, two objects must be created that will eventually be at rest relative to each other, and therefore should be in the same DIG. Once the operation is completed, the code status must be examined to ensure that the objects are indeed in the same DIG.
    - This scenario was completed successfully: The two objects were spheres created with the same X position, so one was directly on top of the the other. When the lower had come to rest relative to the plane, the upper object eventually came to rest on top of the lower object, and they were joined together in one DIG.

- Once there are two objects in a DIG (see above scenario), then another object that is not in the same graph must be created in such a manner that it will disturb the objects in the DIG. The effects of which should be that when the collision is registered, a collision message is sent to each object in the DIG (even if only one was involved in the collision), then the objects in the same graph should be allocated their own graphs, and the collisions resolved.
    - This scenario was completed successfully. The objects were removed from the DIG correctly, and the collisions handled in a believable fashion. The impact of the top two objects was forwarded to the lower object, as it visibly moved after the impact.

- For the issues of scalability, these tests should be repeated when there are more than two objects in a DIG, just to ensure that the messages are passed correctly, and that the collisions are resolved correctly, all within a reasonable frame rate.
    - This scenario was not completed successfully. When, ordinarily, the third object (another sphere with the same X and Z co-ordinates) should have been added to the DIG, a collision was registered which made the lowest object immaterial, and the middle object fell to ground through it. When this object had come to rest, it then became immaterial and the top-most object fell to ground. After this object had bounced once and nearly cleared the object(s) on the ground, an error was reported in the `manipulateDIG()` method

(responsible for allocating objects to the same graph, or taking two objects and putting them in unique graphs) that the graph containing the objects was not found.